Lecture 2

Towards a Verifying Compiler: Logic of Object oriented Programs

Wolfram Schulte
Microsoft Research
Formal Methods 2006

Objects, references, heaps,
Subtyping and dynamic binding,
Pre- and postconditions, method framing

Joint work with **Rustan Leino, Mike Barnett**, Manuel Fähndrich, Herman Venter, Rob DeLine, Wolfram Schulte (all MSR), and *Peter Müller* (ETH), *Bart Jacobs* (KU Leuven) and Bor-Yuh Evan Chung (Berkley)

Review: Boogie PL

Source language
(eg. Spec#)

Translate source language features using particular programming methodology

Intermediate language for verification

BoogiePL

Translate Boogie PL code using particular VC generation

Formulas

Review Boogie PL

- What components does Boogie PL have, and what does it not have?
- What is the purpose of assert, assume and havoc?
- What's the meaning of a procedure and its modifies clause?
- What do we need to translate an OO language into Boogie PL?

Mapping Spec# to BoogiePL

- Axiomatizing Spec#'s class and field declarations
- The storage model
- Translating methods and code
- Method framing (simplified)
- Loop framing

Axiomatizing the Spec# Type System

On notation:

We use the following C# class

```
class C : object {
   object f = null;
   C(){}
 }
```

to describe the result of the axiomatization.

We use the function

Tr (anslate)

to translate Spec# statements into BoogiePL

Axiomatizing the Spec# Type System

Introduce a typename for each Spec# type

```
type C : name;
```

Assert subtyping relationship for program types

```
axiom C <: System.Object;
```

by using a predefined partial order operation <:

Axiomatizing C#' Type Declarations

Introduce field names as constants

```
const C.f : name;
```

Assert field properties (kind, type etc).

```
axiom IsRefField(C.f, System.Object);
```

by using the appropriate functions

```
function IsRefField(field:name, type:name) returns bool
```

Storage Model

Use Boogie’s type ref to denote runtime object references

A Heap maps object references and field names to values

var Heap: [ref, name] any; // Heap : ref × name → any

Allocatedness is represented as another field of the heap

const allocated: name;

Access to an instance field f declared in C is translated as

assert o ≠ null ; x := Heap[o, C.f]

Tr[[x = o.f;]] = assert o ≠ null ; Heap[o, C.f] := x

Tr[[o.f = x;]] =

Allocation

```
Tr[[x = new T()]]   =
        {var o: ref;
        assume o != null ∧ typeof(o) == T;
        assume Heap[o, allocated] == false;
        Heap[o, allocated] := true;
        call T..ctor(o); }
```

Methods

Recall: Boogie PL

- has only procedures, no instance methods
  →Add *this* as first parameter to generated proc
- is weakly typed (just int, bool, ref)
  →Spec# types must be preserved via contracts
- has no idea of heap properties
  →Allocatedness must be preserved via contracts
- has no inheritance
  →Strengthening of postconditions must be implemented via multiple procedures

Constructors and Non-Virtual Methods

```
Tr [[C() {}  ]] =
        proc C..ctor(this: ref);
            requires this != null ∧ typeof(this) <: C;
            modifies Heap;

        impl C..ctor(this: ref)
        {       assume Heap[this, allocated] == true;

                //for constructors only
                assume Heap[this, C.f] == null;
                call System.Object..ctor(this);
                …
        }
```

Preserve
type information

Preserve
initialization
semantics

Virtual Methods: Example

```
class Cell{
    public int x;

   protected virtual void Set(int x)
       modifies this.*;
       ensures this.x == x;
   {   this.x = x; }

   public void Inc(int x)
       modifies this.*;
       ensures this.x==old(this.x)+x;
   {   this.Set(Get()+x); }
}
```

```
class BackupCell: Cell{
    int b;

  protected override  void Set(int x)

    ensures this.b == old(this.x);
  {   this.b = this.x; base.Set(x); }
```

Behavioral Subtyping

Behavioral Subtyping should guarantee *substitutability*

- Wherever an object of type T is expected an object of type S, where S<:T, should do without changing the program's behavior expressed in wp

Sufficient conditions: Let M1 be a virtual method and M2 be its overridden method, then

- M2 can *weaken* M1's *precondition*
- M2 can *strengthen* M1's *postcondition*

Virtual Methods

Translate each *method* m declared in C into a

> proc m.C (this, …) requires this != null $\wedge$ typeof(this) <: C;

…

The precondition of the overriding method is inherited from the overridden method; additional postconditions are conjoined

Translate *calls* of the form o.m() to the method on o's most specific static type

Method Framing

- For sound verification we assume that every method modifies the heap
- *Modifies clauses* in Spec# express which locations (evaluated in the method's prestate) a method is allowed to modify
- Modifies clauses for an object o or array a have the form:
  - o.f allows modification of o's f field
  - o.* allows modification of all of o's fields
  - a[k] allows modification of a's array location k
  - a[*] allows modification of all of a's array locations

Method Framing

Let W denote all locations a method is allowed to modify

- The Boogie PL post condition for a Spec# modifies clause

Tr [[W]] =

$$(\forall o\text{: ref, f: name} :: \text{old}(\text{Heap}[o,\text{allocated}]) \Rightarrow (o,f) \in \text{old}(W) \vee \text{old}(\text{Heap}[o,f]) = \text{Heap}[o,f])$$

Virtual Methods: Example Translation

Spec#

```
protected virtual void Set(int x)
  modifies this.*;
```

Boogie

```
proc Cell.Set(this : Cell, x : int)
      requires this != null ∧ typeof(this) <: Cell;
      modifies Heap;
      ensures (∀o:ref, f: name :: old(Heap[o,allocated])
                          ⇒ o = this ∨ old(Heap[o,f]) = Heap[o,f]);
```

Loop Framing

- Loops might change the heap. Let W denote the set of locations potentially changed by the loop

- For sound verification we havoc the heap. We add as loop invariant the assertion that *fields not written to don't change*

Tr [[W]] =

$$(\forall o : ref,\ f{:}\ name :: Heap_{entry}[o, allocated] \Rightarrow f \in W \lor Heap_{entry}[o,f] = Heap_{current}[o,f])$$

where $Heap_{entry/current}$ denote the entry/current incarnations of the Heap variable in the loop

Summary

Verifying object-oriented programs requires to

- axiomatize the declaration environment
  - to keep enough information around for verification
- decide on a storage model
  - to model updates and framing
- translate the method bodies, paying particular attention to
  - partiality of operations
  - virtual dispatch
  - method and loop frames